



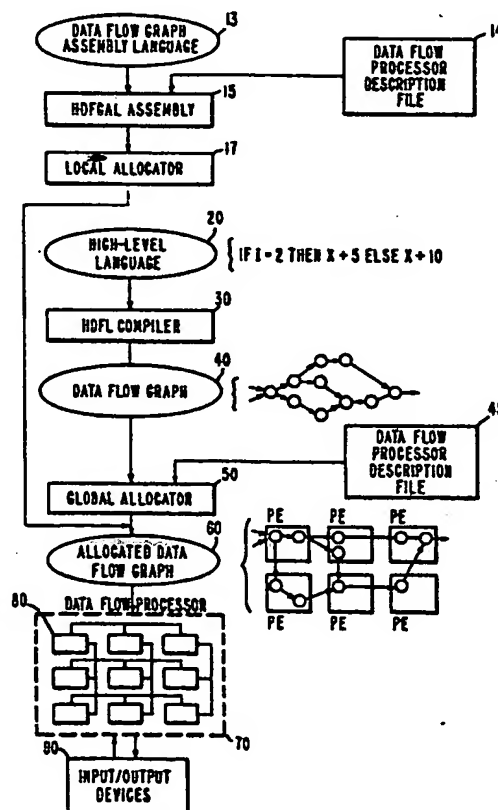
INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification ⁴ : G06F 9/44	A1	(11) International Publication Number: WO 87/ 06034 (43) International Publication Date: 8 October 1987 (08.10.87)
(21) International Application Number: PCT/US87/00410 (22) International Filing Date: 2 March 1987 (02.03.87) (31) Priority Application Number: 847,087 (32) Priority Date: 31 March 1986 (31.03.86) (33) Priority Country: US (71) Applicant: HUGHES AIRCRAFT COMPANY [US/US]; 7200 Hughes Terrace, Los Angeles, CA 90045-0066 (US). (72) Inventors: CAMPBELL, Michael, L. ; 3326 Sawtelle Boulevard, #3, Los Angeles, CA 90066 (US). FINN, Dennis, J. ; 12133 Mitchell Avenue, #102, Los Angeles, CA 90066 (US). TUCKER, George, K. ; 11615 Mississippi, Los Angeles, CA 90025 (US). VAHEY, Michael, D. ; 1727 Ruhland, Manhattan Beach, CA 90266 (US). VEDDER, Rex, W. ; 8162 Manitoba Street, #205, Playa del Rey, CA 90291 (US).	(74) Agents: TAYLOR, Ronald, L. et al.; Hughes Aircraft Company, Post Office Box 45066, Bldg. C1, M.S. A126, Los Angeles, CA 90045-0066 (US). (81) Designated States: DE (European patent), FR (European patent), GB (European patent), IT (European patent), JP. Published With international search report. Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.	

(54) Title: DATA-FLOW MULTIPROCESSOR ARCHITECTURE FOR EFFICIENT SIGNAL AND DATA PROCESSING

(57) Abstract

A data-flow architecture and software environment for high-performance signal and data processing. The programming environment allows applications coding in a functional high-level language (20) which a compiler (30) converts to a data-flow graph form (40) which a global allocator (50) then automatically partitions and distributes to multiple processing elements (80), or in the case of smaller problems, coding in a data-flow graph assembly language so that an assembler (15) operates directly on an input data-flow graph file (13) and produces an output which is then sent to a local allocator (17) for partitioning and distribution. In the former case a data-flow processor description file (45) is read into the global allocator (50), and in the latter case a data-flow processor description file (14) is read into the assembler (15). The data-flow processor (70) consists of multiple processing elements (80) connected in a three-dimensional bussed packet routing network. Data enters and leaves the processor (70) via input/output devices (90) connected to the processor. The processing elements are designed for implementation in VLSI (very large scale integration) to provide realtime processing with very large throughput. The modular nature of the computer allows adding more processing elements to meet a range of throughput and reliability requirements. Simulation results have demonstrated high-performance operation, with over (64) million operations per second being attainable using only 64 processing elements.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria	FR	France	ML	Mali
AU	Australia	GA	Gabon	MR	Mauritania
BB	Barbados	GB	United Kingdom	MW	Malawi
BE	Belgium	HU	Hungary	NL	Netherlands
BG	Bulgaria	IT	Italy	NO	Norway
BJ	Benin	JP	Japan	RO	Romania
BR	Brazil	KP	Democratic People's Republic of Korea	SD	Sudan
CF	Central African Republic	KR	Republic of Korea	SE	Sweden
CG	Congo	LI	Liechtenstein	SN	Senegal
CH	Switzerland	LK	Sri Lanka	SU	Soviet Union
CM	Cameroon	LU	Luxembourg	TD	Chad
DE	Germany, Federal Republic of	MC	Monaco	TG	Togo
DK	Denmark	MG	Madagascar	US	United States of America
FI	Finland				

DATA-FLOW MULTIPROCESSOR ARCHITECTURE FOR EFFICIENT
SIGNAL AND DATA PROCESSING

1 BACKGROUND OF THE INVENTION

1. Field of the Invention

5 The present invention relates to methods and
apparatus for performing high-speed digital computations
of programmed large-scale numerical and logical problems,
in particular to such methods and apparatuses making
use of data-flow principles that allow for highly
parallel execution of computer instructions and
calculations.

10

2. Description of the Technology

15 In order to meet the computing requirements of
future applications, it is necessary to develop
architectures that are capable of performing billions of
operations per second. Multiprocessor architectures are
widely accepted as the class of architectures that will
enable this goal to be met for applications that have
sufficient inherent parallelism.

20 Unfortunately, the use of parallel processors
increases the degree of complexity of the task of
programming computers by requiring that the program be
partitioned into concurrently executable processes and
distributing them among the multiple processors, and
that asynchronous control be provided for parallel
25 process execution and inter-process communication. An
applications programmer must partition and distribute

1 his program to multiple processors, and explicitly
coordinate communication between the processors or
shared memory.

5 Applications programming is extremely expensive
even using current single-processor systems, and is
often the dominant cost of a system. Software develop-
ment and maintenance costs are already very high without
programmers having to perform the additional tasks
described above. High-performance multiprocessor
10 systems for which software development and maintenance
costs are low must perform the extra tasks required for
the programmer and be programmable in a high-level
language.

15 There are different classes of parallel processing
architectures that may be used to obtain high performance.
Systolic arrays, tightly coupled networks of von Neumann
processors, and data flow architectures are three such
classes.

20 Systolic arrays are regular structures of identical
processing elements (PEs) with interconnection between
PEs. High performance is achieved through the use of
parallel PEs and highly pipelined algorithms. Systolic
arrays are limited in the applications for which they
may be used. They are most useful for algorithms which
25 may be highly pipelined to use many PEs whose intercom-
munications may be restricted to adjacent PEs (for
example, array operations). In addition, systolic
arrays have limited programmability. They are "hardwired"
designs in that they are extremely fast, but inflexible.
30 Another drawback is that they are limited to using
local data for processing. Algorithms that would
require access to external memories between computations
would not be suitable for systolic array implementation.

1 Tightly coupled networks of von Neumann processors
typically have the PEs interconnected using a communica-
tion network, with each PE being a microprocessor
having local memory. In addition, some architectures
5 provide global memory between PEs for interprocessor
communication. These systems are most well suited for
applications in which each parallel task consists of
code that can be executed efficiently on a von Neumann
processor (i.e., sequential code). They are not well
10 suited for taking full advantage of low-level (micro)
parallelism that may exist within tasks. When used for
problems with low-level parallelism they typically give
rise to large ALU (arithmetic and logical unit) idle
times.

15 Data flow multiprocessor architectures based on the
data flow graph execution model implicitly provide for
asynchronous control of parallel process execution and
inter-process communication, and when coupled with a
functional high-level language can be programmed as a
20 single PE, without the user having to explicitly identify
parallel processes. They are better suited to taking
advantage of low-level parallelism than von Neumann
multiprocessor architectures.

25 The data flow approach, as opposed to the traditional
control flow computational model (with a program counter),
lets the data dependencies of a group of computational
operations determine the sequence in which the operations
are carried out. A data flow graph represents this
information using nodes (actors) for the operations and
30 directed arcs for the data dependencies between actors.
The output result from an actor is passed to other
actors by means of data items called tokens which
travel along the arcs. The actor execution, or firing,
occurs when all the actor's input tokens are present on

1 its input arcs. When the actor fires, or executes, it
uses up the tokens on its input arcs, performs its
intended operation, and puts result tokens on its
output arcs. When actors are implemented in an archi-
5 tecture they are called templates. Each template consists
of slots for an opcode, operands, and destination
pointers, which indicate the actors to which the results
of the operation are to be sent.

The data flow graph representation of an algorithm
10 is the data dependency graph of the algorithm. The
nodes in the graph represent the operators (actors) and
the directed arcs connecting the nodes represent the
data paths by which operands (tokens) travel between
operands (actors). When all the input tokens to an
15 actor are available, the actor may "fire" by consuming
its input tokens, performing its operation on them, and
producing some output tokens. In most definitions of
data flow a restriction is placed on the arcs and actors
so that an arc may have at most one input token on
20 it at a time. This implies that an actor may not fire
unless all of its output arcs are empty. A more general
definition allows for each arc to be an infinite queue
into which tokens may be placed.

All data flow architectures consist of multiple
25 processing elements that execute the actors in the data
flow graph. Data flow architectures take advantage of
the inherent parallelism in the data flow graph by
executing in separate PEs those actors that may fire in
parallel. Data flow control is particularly attractive
30 because it can express the full parallelism of a problem
and reduce explicit programmer concern with interprocessor
communication and synchronization.

1 In U.S. Patent Number 3,962,706--Dennis et al., a
data processing apparatus for the highly parallel
execution of stored programs is disclosed. Unlike the
present invention, the apparatus disclosed makes use of
5 a central controller and global memory and therefore
suffers from the limitations imposed by such an
architecture.

 U.S. Patent Number 4,145,733--Misunas et al.
discloses a more advanced version of the data processing
10 apparatus described in U.S. Patent Number 3,962,706.
However, the apparatus disclosed still contains the
central control and global memory that distinguish it
from the present invention.

 U.S. Patent Number 4,153,932--Dennis et al.
15 discloses another version of the apparatus disclosed in
the previous two patents, distinguished by the addition
of a new network apparently intended to facilitate
expandability, but not related to the present invention.

 In U.S. Patent Number 4,418,383--Doyle et al. a
20 large-scale integration (LSI) data flow component for
processor and microprocessor systems is described. It
bears no substantive relation to the processing element
of the present invention, nor does it teach anything
related to the data flow architecture of the present
25 invention.

 None of the inventions disclosed in the patents
referred to above provides a processor designed to
perform image and signal processing algorithms and
related tasks that is also programmable in a high-level
30 language which allows exploiting a maximum of low-level
parallelism from the algorithms for high throughput.

1 The present invention is designed for efficient
realization with advanced VLSI circuitry using a smaller
number of distinct chips than other data flow machines.
It is readily expandable and uses short communication
5 paths that can be quickly traversed for high performance.
Previous machines lack the full capability of the present
invention for large-throughput realtime applications
in data and signal processing in combination with the
easy programmability in a high-level language.

10 The present invention aims specifically at providing
the potential for performance of signal processing
problems and the related data processing functions
including tracking, control, and display processing on
the same processor. An instruction-level data flow
15 (micro data flow) approach and compile time (static)
assignment of tasks to processing elements are used to
get efficient runtime performance.

SUMMARY OF THE INVENTION

20 The present invention is a data flow architecture
and software environment for high performance signal and
data processing. The programming environment allows
applications coding in a functional high-level language,
the Hughes Data Flow Language, which is compiled to a
25 data flow graph form which is then automatically
partitioned and distributed to multiple processing
elements. As an alternative for smaller scale problems
and for simulation studies, a data flow graph language
assembler and local allocator allow programming directly
30 in data flow graph form.

 The data flow architecture consists of many
processing elements connected by a three-dimensional
bussed packet routing network. The processing
elements are designed for implementation in VLSI (very
35 large scale integration) to provide realtime processing
with very large throughput. The modular nature of the

1 data-flow processor allows adding more processing
elements to meet a range of throughput and reliability
requirements. Simulation results have demonstrated
high-performance operation.

5 Accordingly, it is one object of the present
invention to provide a data-flow multiprocessor that is
a high-performance, fault-tolerant processor which can
be programmed in a high-level language for large-
throughput signal and data processing applications.

10 It is another object of the present invention,
based on data-flow principles so that its operation is
data-driven rather than instruction-driven, that it
allow fast, highly-parallel computation in solving
complex, large-scale problems.

15 It is a further object of the present invention to
provide for an interconnection network for the multiple
processing elements that is split up into as many
components as there are processing elements, so that the
communication network is equally distributed over all
20 the processing elements, and if there are N processing
elements, each processing element has $1/N$ of the inter-
connection network to support it.

It is yet another object of the present invention
to provide for static allocation (that is, at compile
25 time) of the program to the processing elements.

It is still another object of the present invention
to use processing elements which have been designed for
implementation using only two distinct chips in VLSI
(very large scale integration) to provide realtime
30 processing with very large throughput.

Finally, it is an object of the present invention
that the modular nature of the data-flow processor
allow adding more processing elements to meet a range of
throughput and reliability requirements.

1 An appreciation of other aims and objects of the
present invention and a more complete and comprehensive
understanding of this invention may be achieved by
studying the following description of a preferred
5 embodiment and by referring to the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

10 FIG. 1 is a schematic block diagram of the present
invention, with illustrative information about some of
its parts to the right of the block diagram.

 FIG. 2 is a schematic representation of how the
processing elements are connected together in a three-
dimensional bussed packet routing network.

15 FIG. 3 shows how a data packet consists of a
packet type, a PE and template address, and one or more
data words.

 FIG. 4 illustrates the organization of a processing
element in schematic block diagram form.

20 FIG. 5 shows how templates and arrays are mapped
into physical memories.

 FIG. 6 gives examples of some primitive actors
which are implemented directly in hardware.

25 FIG. 7 shows an example of a compiler generated
data flow graph corresponding to the expression "if y1
<= y2 then y2 * 2 + 1 else y1 + x0 endif".

30 FIG. 8 is a simulation results graph of throughput
for the program radar3na (in millions of instructions
per second) versus the number of processing elements.
The curve marked "A" is for a random allocation
algorithm, the curve marked "B" for an allocation
algorithm using transitive closure, and the curve marked
"C" for an allocation algorithm using nontransitive
closure.

1 FIG. 9 is a plot of simulation results for the
program radarb. The ordinate represents throughput in
MIPS and the abscissa represents number of processing
5 elements. The lower curve is for a random allocation
algorithm and the upper curve is for a nontransitive
closure allocation algorithm.

10 FIG. 10 is a simulation results graph of percentage
of time the ALU is busy versus the number of processing
elements for the program radar3na. The solid curves
marked "D" and "G" are average ALU busy time and
maximum ALU busy time, respectively, for a transitive
closure allocation algorithm. The curves marked "E" and
"F" are average ALU busy time and maximum ALU busy time,
15 respectively, for a nontransitive closure allocation
algorithm.

20 FIG. 11 shows percentage of time the ALU is busy
in a simulation of the program radarb versus number of
processing elements, using a nontransitive closure
allocation algorithm. The lower curve is for average
ALU busy time and the upper curve is for maximum ALU
busy time.

25 FIG. 12 is a graph of percentage of maximum
achieved throughput versus percentage of time the
average ALU is busy. The solid circles are from
simulation results for the program radarb using a
nontransitive closure allocation algorithm. The
x-symbols and open circles are for the program radar3na
using transitive closure and nontransitive closure
allocation algorithms, respectively.
30

1 FIG. 13 is a plot of the percentage of packet
communication that is local (intra-PE as opposed to
inter-PE) versus number of processing elements for the
program radar3na. The lower curve is for a transitive
5 closure allocation algorithm and the upper curve is for
a nontransitive closure allocation algorithm.

10 FIG. 14 is a plot of the percentage of packet
communication that is local (intra-PE as opposed to
inter-PE) versus number of processing elements for the
program radarb using a nontransitive closure allocation
algorithm.

15 FIG. 15 is a graph of the length (in packets) of
the result queue versus number of processing elements
for the nontransitive closure allocation of the program
radarb. The lower curve is average queue length and the
upper curve is maximum queue length.

20 FIG. 16 is a plot of average communication packet
latency (in clock cycles) versus number of processing
elements for nontransitive closure allocation of the
program radarb.

DESCRIPTION OF A PREFERRED EMBODIMENT

25 FIG. 1 is a schematic block diagram of the present
invention, a data flow architecture and software environ-
ment 10 for high-performance signal and data processing.
The programming environment allows applications coding
in a functional high-level language which results in a
program file 20 which is input into a compiler 30 which
30 converts it to a data flow graph form 40 which a global
allocator 50 then automatically partitions and distributes
to multiple processing elements 80. In the case of smaller
problems, programming can be done in a data flow graph
and assembled by an assembler 15 that operates directly
35 on an input data flow graph file 13 whose output is

1 then sent to a local allocator 17 for partitioning and
distribution. In the former case a data flow processor
description file 45 is read into the global allocator 50
and in the latter case a data flow processor description
5 file 14 is read into the assembler 15. The data flow
processor 70 consists of many processing elements 80
connected in a three-dimensional bussed packet routing
network. Data enters or leaves the processor 80 by
means of input/output devices 90 connected to the
10 processor.

The Three-Dimensional Bussed Network

As shown in FIG. 2, the data flow processor 70
comprises 1 to 512 identical processing elements
15 connected by a global inter-PE communication network.
This network is a three-dimensional bussed network in
which the hardware implements a fault-tolerant store-
and-forward packet-switching protocol which allows any
PE to transfer data to any other PE. Each processing
20 element contains queues for storing packets in the
communication network, and the appropriate control for
monitoring the health of the processing elements and
performing the packet routing.

In the three-dimensional bussed interconnection
25 network not all processing elements are directly
connected, so a store-and-forward packet routing
technique is used. This algorithm is implemented in the
communication chip 81, which connects a processing chip
80 to the row, column, and plane buses 82,84,86. The
30 communication chip 81 acts like a crossbar in that it
takes packets from its four input ports and routes them
to the appropriate output ports. In addition, it
provides buffering with a number of first-in-first-out
queues, including the processor input queue 112 and
35 processor output queue 114.

1 The three-dimensional bussed network is optimized
for transmission of very short packets consisting of a
single token. As shown in FIG. 3, each packet consists
of a packet type, an address, and a piece of data.
5 Different types of packets include normal token packets,
initialization packets, and special control packets for
machine reconfiguration control. The address of each
packet consists of a processing element address and a
template address which points to one particular actor
10 instruction within a processing element. The data can
be any of the allowed data types of the high-level data
flow language or control information if the packet is a
control packet.

15 A configuration of up to $8 \times 8 \times 8$ or 512 processing
elements can be physically accommodated by the communi-
cation network. Many signal processing problems could
potentially use this many processing elements without
overloading the bus capacity because of the ease of
partitioning these algorithms. However, for general
20 data processing the bus bandwidth will start to saturate
above four processing elements per bus. More processing
elements can be added and performance will increase but
at lower efficiency per processing element.

25 A single-path routing scheme is used in transferring
packets between PEs. In other words, the same path is
used every time packets are sent from a given source PE
to a given sink PE. This guarantees that packets sent
from an actor in the given source PE to an actor in the
given sink PE arrive in the same order in which they
30 were sent, which is necessary when actors are executed
more than once (as, for example, when the graph is
pipelined).

1 Each PE continually monitors its plane, column, and
row buses looking for packets it should accept. PEs
accept packets addressed directly to them and packets
that need to be rerouted to other PEs through them. For
5 example, if a packet is put on a plane bus, all PEs on
that bus examine the packet address and the PE whose
plane address matches the packet's plane address accepts
the packet.

10 Fault Tolerance

The communication network is designed to be reliable,
with automatic retry on garbled messages, distributed
bus arbitration, alternate-path packet routing, and
failed processing element translation tables to allow
15 rapid switch-in and use of spare processing elements.

Static fault tolerance is fully supported. Upon
determination of a PE failure, a spare PE can be loaded
with the templates from the failed PE and operation
continued. This creates two problems, however: 1) the
20 spare PE has a different address than the PE it replaced,
and 2) messages that were to be routed through the
failed PE must instead be routed around it.

The first problem is solved by two methods. In the
long term (days to months) the applications program can
25 be reallocated using the allocator software during a
scheduled maintenance period. For immediate recovery
(several seconds), a small number of failed PE address
translation registers, called the error memory 110, are
provided in each PE. When a PE fails, its address is
30 entered in the error memory 110 followed by the address
of its replacement PE. Each packet generated is checked
against the error memory and if a match is made, the
replacement address is substituted for the address of
the failed PE.

1 Routing of packets around failed PEs is accomplished
by each PE keeping track of which directly connected
PEs are operative and which have failed. In the case
of failed PEs the sending PE routes packets to alternate
5 buses.

Dynamic fault tolerance can be provided by running
two or more copies of critical code sections in parallel
in different PEs and voting on the results. Unlike
difficulties encountered in other types of parallel
10 processors, the data flow concept avoids synchronization
problems by its construction, and interprocess communi-
cation overhead is minimized because it is supported in
hardware. This software approach to dynamic fault
tolerance minimizes the extra hardware required for
15 this feature.

Packets

The packets that are transferred contain either
16-bit or 24-bit token values (see FIG. 3). The data
20 paths are 17 bits wide: 16 data bits plus 1 tag bit.
Each packet contains six type bits, a PE address, an
actor address, and the data being transmitted from one
actor to another. The PE address identifies the
destination PE and the actor address identifies the
25 actor within that PE to which data is being sent. The
PE address is 9 bits (3 bits for each plane, column, and
row address) and can be used to address up to 512
distinct PEs (such as there would be in an 8x8x8 cubic
arrangement of PEs). Variable-length packets are
30 supported by the network protocol, with the last word of
a packet transmission indicated by an end-of-packet bit.

The Processing Element

Each processing element 80 consists of a communications chip 81, a processing chip 120, and memory as shown in FIG. 4. The communication network is distributed over all the PEs for improved fault tolerance. The part of the communications network associated with a single PE is represented in FIG. 4 by external plane, column, and row buses 82, 84, 86. The external buses 82,84,86 use parity, a store-and-forward protocol, and a two-cycle timeout that indicates a bus or PE failure if the packet reception signal is not received within two cycles. The parity and timeout features are used for error detection. The store-and-forward protocol is necessary because the input queue at the receiving communication chip may be full, in which case the sending communication chip needs to retransmit the packet later. The arbitration control of the external buses 82,84,86 is decentralized for high reliability. Pairs of input/output queues 88,100; 102,104; and 106,108 are used to buffer the data entering or leaving via the external plane, column, and row buses 82,84,86. Two internal buses 89 and 107 are used for sending packets from the processing chip plane, column, and row input queues 88,102,106 to the processor plane, column, and row output queues 100,104,108. All of the buses use round robin arbitration.

The communication chip 81 accepts tokens addressed to actors stored in its associated processing chip 120 and passes them to it. An error memory 110 in the communications chip 81 contains a mapping of logical PE addresses to physical PE addresses. Normally the two are the same, but if a PE fails, its logical address is mapped to the physical address of one of the spare PEs.

1 Static fault tolerance is used. When a PE fails, self-
test routines are used to determine whether the failure
is temporary or permanent. If it is permanent, the code
that was allocated to the failed PE must be reloaded
5 into the spare PE that will have the address of the
failed PE. The program must then be restarted from the
last breakpoint. The communication chip is highly
pipelined so that it can transmit a word of a packet
almost every cycle. About 5 to 6 million packets per
10 second can be transmitted by the communication chip.

This arrangement for the processing element was
chosen to simplify VLSI design. There were only two
VLSI chip types to design, and partitioning the PE into
the communication chip 81 and the processing chip 120
15 minimized the number of input/output pins per chip.
Both chip types are custom designed VHSIC 1.25 micron
CMOS/SOS chips operating at 20 MHz.

Each individual PE is a complete computer with its
own local memory for program and data storage. Associated
20 with each processing chip are two random access memories
(RAMs) 146 and 156 that store the actors allocated to
the PE. These two memories, the destination memory 146
and template memory 156, are attached to processing
chip 120. Each is composed of multiple RAM chips and
25 has an access time of less than 80 nanoseconds, with
two cycles required per memory access. A single bi-
directional bus 158 is used to communicate between the
communication chip 81 and the processing chip 120.

The processing chip contains four special-purpose
30 microprocessors which are choose to call "micromachines".

The processing chip 120 accepts tokens from the
communications chip 81 and determines whether each token
allows an actor to fire. If not, the token is stored
until the matching token or tokens arrive. If a token

1 does enable an actor, then the actor is fetched from
memory and executed by an ALU micromachine 144 in the
processing chip 120. The resulting value is formed
into one or more tokens and they are transmitted to
5 the other actors that are expecting them.

The memory attached to each processing chip is used
to store actors represented as templates. A template
consists of a slot for an opcode, a destination list of
addresses where results should be sent, and a space for
10 storing the first token that arrives until the one that
matches it is received.

The memory is also used to store arrays, which can
be sent to the memory of a single processing element or
distributed over many processing elements. With dis-
15 tributed arrays, it is possible for an actor executing
in one processing element to need access to an array
value stored in the memory of another processing element.
Special actors are provided in the architecture for
these nonlocal accesses. Given the array index or
20 indices, the address of the processing element containing
the value is calculated based on the way the array is
distributed, and a request for the value is sent via
the communication network. The other processing element
then responds by sending back the requested value as a
25 normal token. Nonlocal array updates are handled
similarly.

The processing chip is a pipelined processor with
the following three operations overlapped: 1)
instruction/operand fetch and data flow firing rule
30 check, 2) instruction execution, and 3) matching results
with destination addresses to form packets. There is
some variance in the service times of each of these
units for different instructions, so queueing is
provided between the units as shown in FIG. 4.
35

1 The instruction fetch and data flow firing rule
check is performed by two parallel micromachine units,
the template memory controller 130 and the destination
memory controller 122. The templates are spread across
5 three independent memories: the fire detect memory 132,
the template memory 156, and the destination memory
146. The first 4K locations of each of these memories
contain addresses of actors. The fire detect memory
132 only has 4K locations. The template memory 156 and
10 destination memory 146 have additional memory that is
used to store variable-length data associated with each
actor, array data, and queue overflow data. The templates
are split between the three memories so that the template
memory controller 130 and destination memory controller
15 122 can operate in parallel and thus prepare actors for
firing more quickly than if one memory and one controller
were used.

 When a packet arrives at the processing chip, the
status of the template to which the packet is addressed
20 is accessed from the fire detect memory 132 and a
decision is made on whether the template is ready to
fire. The status bits are stored in the on-chip fire
detect memory 132 to allow fast access and update of
template status. If the template is not ready to fire,
25 the arriving token (operand) is stored in the template
memory 156.

 If the template is ready to fire, the template
memory controller 130 fetches the template opcode and
operand stored in the template memory 156, combines them
30 with the incoming operand, which enabled the actor to
fire, and sends them to the firing queue 138, from which
the arithmetic and logic unit (ALU) micromachine 144
will fetch them. Simultaneously, the destination
memory controller 122 begins fetching the destination
35

1 addresses to which the template results should be sent
and stores these addresses in the destination queue
134. Since each result of each template (actor) may
need to be sent to multiple destinations, the destination
5 memory 146 includes an overflow storage area to accommodate
lists of destinations for each result of each actor.
FIG. 5 shows how templates and arrays are mapped into
physical memories.

The results of the actor execution performed in the
10 ALU micromachine 144 are put into the result queue 142.
The results in the result queue 142 and the destinations
in the destination queue 134 are combined together into
packets by the destination tagger micromachine 136 and
sent back to the template memory controller 130 (via
15 the feedback queue 138) or to other PEs (via the to-
communication queue 124).

To summarize, the four main functions of a proces-
sing element are communication network processing,
actor fire detection, actor execution, and result token
20 formation. All four of these functions are performed
concurrently in a pipelined fashion.

A stand-alone processing element is capable of
performing 2 to 4 micro-operations per seconds (MOPS)
depending on the instruction mix used. In this case an
25 MOP is defined as a primitive actor instruction; these
vary in complexity from a simple 16-bit add which is
completed in one micro-instruction to some array adres-
sing instructions which take over ten cycles, or a
16-bit divide which takes approximately 25 cycles.

30

35

1 Two separate memory interfaces 148,150 and 152,154
allow a large memory-processor bandwidth which is
necessary to sustain high performance. The design goal
of minimizing chip types and power consumption resulted
5 in a simple design for the ALU: there is no hardware
multiply -- multiplication is performed by a modified
Booths algorithm technique. Each of the chips has less
than 150 pins, consists of approximately 15K gates,
and operates at a clock rate of 20 MHz.

10

Software Environment

The preferred embodiment of the present invention
is programmed in the Hughes Data Flow Language (HDFL),
which is a high-level functional language. A record of
15 the HDFL program 20 is read into the compiler 30 which
translates it into a parallel data flow graph form 40
which along with a description of the processor configura-
tion 45 is fed into the global allocator 50 for distri-
bution to the multiple processing elements 80. The
20 allocator employs static graph analysis to produce a
compile-time assignment of program graph to hardware
that attempts to maximize the number of operations which
can proceed in parallel while minimizing the inter-PE
communication.

25 Since one of the primary goals of the present
invention is to provide a high-level language programming
capability in order to reduce software cost, a high-level
language had to be found in which the parallelism
inherent in many problems could be easily expressed.
30 Current sequential languages such as FORTRAN and Pascal
were eliminated because of their inherent lack of
parallelism. Ada and other multi-tasking languages
were rejected because they require explicit programmer
concern with creating and synchronizing multiple tasks,

35

1 which adds complexity and cost to software development.
Within a specific process, these languages are also
subject to the same lack of parallelism as the FORTRAN-
class languages. It was determined that an applicative
5 data flow language such as VAL (see "The VAL Language:
Description and Analysis," by J.R. McGraw, in ACM
Transactions on Programming Languages and Systems,
Volume 6, Number 1, January 1982, pages 44 through 82)
or Id (see "The (Preliminary) Id Report," by Arvind
10 et al., Department of Computer Science, TR114a, University
of California, Irvine, May 1978) was needed to allow an
effective extraction of parallelism and efficient
mapping to multiprocessor hardware. This led to the
development of the Hughes Data Flow Language, which is
15 a general purpose high-level programming language for
data flow computers. The HDFL is designed to allow
full expression of parallelism. It is an applicative
language but includes the use of familiar algebraic
notation and programming language conventions.

20 The Hughes Data Flow Language is value oriented,
allowing only single-assignment variables. Its features
include strong typing, data structures including records
and arrays, conditionals (IF THEN ELSE), iteration
(FOR), parallel iteration (FORALL), and streams. An
25 HDFL program consists of a program definition and zero
or more function definitions. There are no global
variables or side effects; values are passed via
parameter passing.

30 Immediately below is shown a simple illustrative
example of HDFL:

```
type xy = record [ x: integer; y: integer ];
```

```
1  constant scalefactor = 2; %This is a comment.

function foo( xyvar:xy; x0,y1,y2:integer returns xy,
              integer)

5  constant offset = 1;

result:
    xy( xyvar.x + x0, xyvar.y + y1) ,
10  if y1 > y2    %Either branch creates a single value,
    then y2 * scalefactor + offset
    else y1 + x0
    endif
endfun

15
```

The example shown immediately above consists of a function "foo" which takes four parameters (one record and three integers), and returns one record and one integer. "Result" is a keyword beginning the body of a function and "endfun" terminates it. The function body consists of a list of arbitrarily complex expressions separated by commas with one expression per return value. In this example the first expression in the function body is a "record type constructor" which assigns values to the fields of the record result. The conditional below it evaluates to an integer value. Constants and types may be declared before the function header or before the body. Functions may be nested.

30-

35

1 HDFL Compiler

 The compiler translates HDFL into a data flow graph intermediate form composed of primitive data flow actors. Operation proceeds in three phases: 1) syntax
5 checking and parse tree construction, 2) semantics checking and augmentation, and 3) code generation. Each phase is table driven. Following table-driven code generation is a final post-processing stage to eliminate unnecessary code, evaluate constant subgraphs, and
10 perform some optimizations. The graph intermediate form generated by the compiler includes syntactic information and other information which is used by the allocator.

 The primitive actors are those supported directly by the hardware. Some of the actors are in 16-bit form
15 and others are in 32-bit form. Many are simple arithmetic and Boolean actors such as ADD, others are control actors such as ENABLE and SWITCH, or hybrids like LE5, some are used in function invocation such as FORWARD, and others are used for array and stream
20 handling. FIG. 6 shows some of the primitive actors implemented directly in hardware.

 For each construct in the high-level language the compiler has a corresponding data flow graph made up of primitive actors that implements that function. For
25 example, the data flow graph generated from the HDFL conditional expression "if $y_1 \leq y_2$ then $y_2 * 2 + 1$ else $y_1 + x_0$ endif" is shown in FIG. 7. The "then" and "else" branches of the conditional are merged together by sending tokens on these arcs to the same location;
30 this is indicated by merging the output arcs together. Note also that the LE5 actor has some stub output arcs which are not used. The ENABLE actor is present so that when the result of the expression is generated this guarantees that all actors in the graph have fired and
35 the graph is available for further use if desired.

1 The Allocator

 The assignment of actors to processing elements can
have a large impact on the performance of the multi-processor.
For example, since each PE is a sequential computer, actors that
5 potentially can fire in parallel cannot do so if they are
assigned to the same PE. Performance can also be affected by data
communication delays in the inter-PE communication network. It
takes many more clock cycles to transmit a token from one PE
10 to another than it does to transmit a token from one actor
to another in the same PE, which bypasses the communication
network completely.

 This leads to three goals for efficient allocation:
1) minimize inter-PE communication by assigning actors
15 connected in the graph to the same processing element,
2) maximize use of the parallelism of the graph by assigning
actors that can fire in parallel to separate processing
elements, and 3) balance the computational load as evenly as
20 possible among the processing elements.

 In concert with the development of the data flow
architecture and the high-level language compiler 30, an
allocation algorithm was implemented. To begin with in
getting simulation results, a smaller-scale version
25 called the local allocator 17 was implemented.

30

35

1 The Local Allocator

 The input to the local allocator 17 is a file 13
containing a data flow graph in the form of a sequence
of templates. Each template lists the opcode of the
5 operator it represents and the data dependency arcs
emanating from it. This file also lists arrays, each of
which will be assigned to a single processing element or
distributed over many processing elements. A file 14
describing the configuration of the data flow multi-
10 processor 70 to be allocated onto is also read into
the local allocator 17, specifying how many processing
elements 80 there are in each dimension of the three-
dimensional packet routing network connecting the PEs.
For simulation purposes the output of the local
15 allocator 17 consists of two files. The first file
specifies the mapping of each actor of the graph to a
memory location in one of the processing elements, and
the second file specifies how arrays have been assigned
to specific blocks of memory in one or more processing
20 elements. These files can then be read into the
architectural simulator to initialize the simulated
machine.

 The local allocator 17 begins by topologically
sorting the actors of the graph using a variation of
25 breadth-first search (for a description see The Design
and Analysis of Computer Algorithms, by Aho et al.,
published by Addison-Wesley, 1974). In topologically
sorted order, the actors that receive the inputs of the
graph are first, followed by the actors that receive
30 arcs from the first actors, and so on. (For this
purpose we can ignore cycles in the graph by disregarding
back arcs to previously seen actors.) The next step is
to compute the transitive closure of the data flow
graph, which is defined in the discussion of heuristics
35 below.

1 The local allocator then sequentially processes the
sorted list of actors in the graph and assigns each
actor to one of the processing elements. To choose the
best PE for an actor, the algorithm applies several
5 heuristic cost function to each of the PEs, takes the
weighted sum of the results, and uses the PE with the
lowest cost. These heuristics are the heart of both the
local allocator 17 and the global allocator 50.

 Currently three basic heuristics are implemented:
10 communication cost, array-access cost, and parallel
processing cost. The communication and array-access
cost functions correspond to the goal of minimizing
inter-PE communication traffic, and the parallel
processing cost function corresponds to the goal of
15 maximizing parallelism.

 The communication cost function takes an actor and
a PE and returns an approximate measure of the traffic
through the network which would result if from assigning
the given actor to the given PE. In general, when two
20 actors are connected, the further apart they are
allocated, the higher the communication cost.

 The heuristic function uses a distance function to
determine how far apart PEs are in the three-dimensional
bussed communication network. For example, if two PEs
25 are on a common bus, then the distance between them is
"one hop," because a token will have to traverse one bus
to travel from one to the other. The distance between a
PE and itself is zero hops, because the communications
network can be bypassed in transmitting a token.

30

35

1 Because the actors are assigned in topologically
sorted order, when an actor is about to be allocated
most of the actors that it receives input tokens from
have already been allocated. Using the distance
5 function between PEs, the communication cost function
determines how far through the communication network
each input token would have to travel if the actor were
assigned to the given PE. The value of the communication
cost function is just the sum of these distances.

10 The processing cost heuristic uses the transitive
closure of the data flow graph to detect parallelism.
The transitive closure of a directed graph is defined to
be the graph with the same set of nodes and an arc from
one node to another if and only if there is a directed
15 path from one node to another in the original graph. In
the worst case this computation requires time proportional
to the cube of the number of nodes (actors).

 Transitive closure is closely related to parallelism
in data flow graphs, because two actors can fire in
20 parallel unless there is a directed path from one to
the other in the graph, which would force them to be
executed sequentially. Thus, two actors can fire in
parallel unless they are directly connected in the
transitive closure of the graph. This fact is used in
25 the parallel processing cost heuristic to determine
which actors should be assigned to separate PEs in order
to maximize the parallelism of the allocated graph. It
simply assigns a higher cost when potentially parallel
actors (according to the transitive closure) are
30 assigned to the same PE.

1 The local allocator attempts to allocate actors
that access an array close to the array, guided by the
array-access cost function. This heuristic function is
a generalization of the communication cost. It measures
5 the traffic through the network which would result from
assigning a given actor that accesses an array to a
given processing element, depending on how far away the
array is assigned.

10 The local allocator allocates each array to one or
more PEs using similar heuristics. For small arrays
with a small number of actors that access it, the local
allocator will choose to confine the actor to a single
PE in order to speed up access time. If an array is
15 large and has a large number of actors that can access
it in parallel according to the transitive closure, the
program will attempt to distribute the array over many
PEs. The actors that access the array will also be
distributed over these PEs to decrease contention for
access to the arrays.

20

The Global Allocator

25 The global allocator combines the heuristic
approach from the local allocator with a divide-and-
conquer strategy, enabling it to operate on large
graphs. Like the local allocator, it accepts a data
flow graph and information about the dimensions of the
processor. It also accepts a hierarchical representation
of the syntactic parse tree from the first pass of the
30 compiler to guide the allocator as it partitions the
graph into parallel modules. By integrating the compiler
and the allocator in this way, the allocator is able to
take advantage of the way the high-level programmer
chose to partition the writing of the program into
functions, subfunctions, and expressions. We refer to
35 this as "syntax-directed partitioning."

1 The divide-and-conquer strategy reduces the
problem to two related subproblems: partitioning the
input graph into a set of smaller, more tractable
modules, and heuristically assigning each module to a
5 set of processing elements. The algorithm proceeds from
the top down by partitioning the graph into several
modules and assigning each module to some set of the
processing elements of the data flow processor. Then,
recursively it further partitions each module into
10 submodules and assigns each of them to a subset of PEs
within the set of PEs previously to which that module
was previously assigned. This partition-and-assign
process is repeated hierarchically until the individual
submodules are small enough to be allocated efficiently,
15 one actor at a time, to individual PEs.

 The nodes of the parse tree from the compiler
correspond to the syntactic elements of the program such
as functions, subfunctions, loop-bodies, and so forth.
The tree is connected by pointers to the data flow graph
20 so that the actors of the graph become the leaves of the
tree. There is a natural correspondence between nodes
of the parse tree and modules in the data flow graph.
The set of actors below a given node of the tree form
the module of the data flow graph that computes the
25 value of the expression corresponding to that node. For
example, the root of the tree corresponds to the module
consisting of the entire data flow graph program. The
children of the node of the tree correspond to the
subfunctions and subexpressions of the parent node.

30

35

1 The task of partitioning the data flow graph into a
set of modules is guided by this syntactic parse tree.
The global allocator partitions a module corresponding
to an expression into a set of submodules corresponding
5 to the subexpressions of the expression. In terms of
the syntactic parse tree, it splits up a node into the
children of the node. In a functional data flow
language, expressions and functions can generally be
computed in parallel because there are no side effects.
10 Therefore these syntactic elements are usually ideal
choices in the partitioning of the corresponding data
flow graph.

 However, these modules are usually not completely
parallel; there can be some data dependencies between
15 them. For example, if there is an expression in the
data flow language program that is assigned to a value
name, then there will be a data dependency from the
module computing that expression to any other modules
that refer to that value name. The global allocator
20 finds such data dependencies between modules by looking
for data dependency arcs between individual actors in
different modules. These dependencies are then used to
construct a graph called by the "module graph," the
nodes of which correspond to modules of the partitioned
25 data flow graph, and the arcs of which indicate the data
dependencies between submodules. It is essentially
another data flow graph.

 The task of assigning the nodes (submodules) of the
module graph to sets of PEs is similar to the assignment
30 performed by the local allocator program. A variant of
that algorithm is used. First the nodes of the module
graph are topologically sorted, then its transitive
closure is computed. In this way it is never required
to compute the transitive closure of the entire graph at
35 one time, so the inefficiency of the local allocator for
large graphs is avoided.

1 In the global allocator the assignment of modules
(and individual actors) to PEs is guided by two of the
heuristic cost functions defined previously in the
section dealing with the local allocator. They have
5 been generalized to apply to modules consisting of many
individual actors being assigned to sets of PEs. For
the communication cost function the distance function
between PEs is generalized to an average distance
between sets of PEs by using the distances between the
10 individual PEs divided by the number of PEs. For the
generalized parallel processing cost function a higher
cost is assigned whenever parallel modules (according to
the transitive closure of the module graph) are assigned
to intersecting sets of PEs, weighted by the number of
15 PEs in the intersection.

Simulation Results

20 The two programs which have been simulated most
extensively are related to realtime radar signal
processing applications. Both programs have been
simulated using a variety of allocation algorithms and
processing element configurations.

25 The radar3na program has 96 actors, 152 arcs, 17
constants, an average ALU execution time of 7.19 cycles
(50 ns cycle time), an average actor fanout (the number
of output arcs for an actor) of 1.58 arcs, and a degree
of parallelism of 21.14 actor firings per cycle (the
average number of actors which can fire in parallel on
the instruction-level simulator).

30 The radarb program uses a 16-point fast Fourier
transform (FFT) with complex arithmetic. It has 415
actors, 615 arcs, 71 constants, an average ALU execution
time of 4.92 cycles, an average actor fanout of 1.56
arcs, and a degree of parallelism of 80.63 actor firings
35 per cycle.

1 Both programs were simulated on 1x1x1, 2x1x1,
2x2x1, 2x3x1, 2x2x2, 2x2x3, 3x3x2, 3x3x3, 4x3x3, 4x4x3,
and 4x4x4 configurations of processing elements. Radarb
was also simulated on an 8x4x4 configuration. Both of
5 these programs were designed to be continuously processing
incoming data. In the simulations eight sets of data
were used for each program execution. Each input actor
grabbed successive data as soon as it could fire, thus
the programs were processing several sets of input data
10 simultaneously. No explicit pipeline stages existed,
nor were any acknowledgement tokens used to prevent
sets of input data from interfering with each other.
Instead, operand queues were used to guarantee safety.
All three allocation algorithms were used for radar3na,
15 but only the nontransitive closure and random algorithms
were used for radarb because the transitive closure
algorithm would have consumed too much CPU time. In
all, more than 300 simulations were carried out using
these two programs.

20 FIGS. 8 and 9 illustrate that both radar3na and
radarb have significantly better throughput using the
nonrandom allocations. The transitive closure algorithm
yields about the same maximum throughput as the
nontransitive closure algorithm, but uses fewer PEs,
25 because it is more likely than the nontransitive closure
algorithm to place two actors into the same PE when they
fire sequentially. In comparing the results shown in
FIG. 9 with the data flow simulation results of Gostelow
and Thomas (in figures 9d and 9e of their paper "Perfor-
30 mance of a Simulated Data Flow Computer," published in
the Proceedings of the Institute of Electrical and
Electronics Engineers TOC, Volume C-29, Number 10,
October 1980, pages 905-919), it can be seen that the
present invention has a much greater improvement in
35 performance as additional processing elements are used.

1 FIGS. 10 and 11 show how the percentage of time the
ALU is busy executing actors varies with respect to the
number of processing elements. The more PEs that are
used the less time the average ALU is busy. This is due
5 primarily to each PE having fewer actors that are ready
to fire at any given time. It is not due to an increase
in packet (token) latency (see FIG. 16). Note that the
difference between the average ALU busy time and the
maximum ALU busy time is small, which implies that the
10 workload is fairly evenly distributed. In addition,
FIG. 10 shows that the transitive closure and nontransi-
tive closure graphs have similar performance. The
portion of the nontransitive closure graph beyond 20
PEs is not of interest because the throughput does not
15 increase when more than 20 PEs are used.

FIG. 12 shows how FIGS. 8 through 11 imply that
there is a tradeoff between maximizing throughput and
efficiently using PEs. For configurations with very few
PEs, the average ALU is very busy, but the program
20 throughput is significantly less than the maximum that
may be obtained because not all of the parallelism of
the program is being exploited. As more PEs are used,
the program throughput increases but the percentage of
time that the average ALU is busy decreases.

25 This is not to say that allocations of programs may
not be found that achieve high throughput and keep the
average CLU very busy. The relationship between maximiz-
ing throughput and efficiently using PEs depends on the
parallelism in the graph, the allocation, and the
30 hardware. For example, an allocation may have only five
actors that can fire in parallel on the average, but ten
actors that can fire in parallel at a particular time.
In this case the achievement of maximum throughput could
require using ten PEs, even though only five are needed
35 on the average.

1 FIGS. 13 and 14 shows how the percentage of packet
communication that is local (within a PE rather than
between PEs) varies with the number of PEs for radar3na
and radarb. They show that as the number of PEs increases,
5 less of the packet communication traffic is local. As
one might expect, the transitive closure allocation
algorithm has more local packet communication than the
nontransitive closure algorithm. What is surprising is
that for radarb, which has more than four times as many
10 actors as radar3na, the percentage of local packet
communication does not decrease very rapidly, and in
fact sometimes increases, as more PEs are used.

FIG. 15 illustrates the average and maximum length
of the result queue for the nontransitive closure
15 allocation of radarb. Not shown in FIG. 15 because of
the scale chosen are results of 103 and 158 for the
average and maximum queue lengths for one PE, and 42 and
74 for the average and maximum queue lengths for two
PEs. Note that the average queue length decreases
20 rapidly beyond a few PEs, and that for eight or more PEs
the average queue length is less than one packet. This
is characteristic of the other queues in the communica-
tions and processor chips, and indicates that the queue
lengths may be limited to a few words so long as a
25 queue overflow area is provided or other methods are
used to prevent deadlock.

One of the first things that was learned through
simulation was that deadlock was much more likely to
occur in our original architecture. Radar3na and radarb
30 both suffered intra-PE deadlock (firing queue, result
queue, and feedback queues filled up) when allocated to
one PE, and radarb suffered inter-PE deadlock (communi-
cation chip queues, firing queue, and result queues

1 filled up) when allocated to four PEs. Our original
architecture had limited all queues in the communication
and processing chip to a length of eight words. In
order to alleviate both intra-PE and inter-PE deadlock
5 we added a result queue overflow area to the destination
memory that could be used when the portion of the
result queue on the processing chip was full. This
explains the large average and maximum result queue
lengths for configurations with few PEs.

10 FIG. 16 shows how the average communication packet
latency varies with number of PEs. This measure of
latency includes the packet delays encountered in the
communication chips and in accessing the communication
chips. It does not take into account the delays
15 encountered in the template memory controller, firing
queue, ALU, result queue, or destination tagger. It
measures the latency from the output of the DT to
the input of the template memory controller. It is a
good measure of the efficiency of the communication
20 system. Note that for few PEs there is very little
communication chip activity, hence the packet latency
contributed by the communication chip is low. As shown
in FIG. 16 the average communication packet latency
peaks at four PEs and decreases rapidly for more PEs.
25 For 18 or more PEs the average packet latency is almost
constant, implying that the decrease in the rate of
increase of the throughput of radar as more PEs are
used (see FIG. 9) is due primarily to a limited amount
of parallelism in the graph rather than increased
30 communication latency.

1 Although the present invention has been described
in detail with reference to a particular preferred
embodiment, persons possessing ordinary skill in the art
to which this invention pertains will appreciate that
5 various modifications may be made without departing from
the spirit and scope of the invention.

10

15

20

25

30

35

CLAIMSWhat is Claimed is:

- 1 1. A method of data-flow multiprocessing for
highly efficient data and signal processing, including
the steps of:
 writing a program of instructions in a high-
5 level data-flow language onto a storage medium;
 reading said program of instructions from
said storage medium into a compiler;
 compiling said instructions by translating
said instructions into a plurality of machine instructions;
10 inputting a file describing data-flow processor
into a global allocator program;
 running said global allocator program in
order to process said plurality of machine instructions
in order to assign said machine instructions to a
15 plurality of processing elements in said data-flow
processor for execution of said machine instructions;
 inputting a plurality of data into said data-
flow processor in order to execute said program in said
data-flow processor; and
20 executing said machine instructions in said
data-flow processor.
- 1 2. A method of data-flow multiprocessing for
highly efficient data and signal processing, including
the steps of:
 writing a program of instructions in a high-
5 level data-flow language onto a storage medium;
 reading said program of instructions from
said storage medium into a compiler;
 assembling said instructions by translating
said instructions into a plurality of machine instructions;

10 inputting said machine instructions into a
 local allocator program;
 running said local allocator program in order
 to process said plurality of machine instructions in
 order to assign said machine instructions to a plurality
15 of processing elements in said data-flow processor for
 execution of said machine instructions;
 inputting a plurality of data into said data-
 flow processor in order to execute said program in said
 data-flow processor.
20 executing said machine instructions in said
 data-flow processor.

1 3. Data-flow apparatus for highly efficient data
 and signal processing, comprising:
 a compilation means for translating instructions
 written in a high-level data-flow language into a
5 plurality of machine instructions;
 a first input means for communicating programs
 written in said high-level data-flow language to said
 compilation means;
 a data-flow processing means for operating on
10 a plurality of machine instructions;
 said data-flow processing means further
 including:
 a plurality of data-flow processing
 elements, each including:
15 a communication part, a processor
 part, and a plurality of memories; and
 a three-dimensional bussed packet routing
 network including a plurality of
 communications buses connecting said
20 processing elements; and

a global allocation means for accepting a plurality of outputs from said compilation means and for accepting a file of instructions describing said data-flow processing means;

25 a second input means coupled to said data-flow processing means in order to communicate a plurality of data to said data-flow processing means; and

 a plurality of output means coupled to said data-flow processing means in order to communicate a
30 plurality of results from said data-flow processing means to an output terminal means.

1 4. Data-flow apparatus for highly efficient data and signal processing, comprising:

 an assembling means for translating instructions written in a high-level data-flow graph language into a
5 plurality of machine instructions;

 a first input means for communicating programs written in said high-level data-flow graph language and for communicating a file of instructions describing a data-flow processing means to said assembling means;

10 a data-flow processing means for operating on a plurality of machine instructions;

 said data-flow processing means further including:

 a plurality of data-flow processing
15 elements, each including a communications part, a processor part, and a plurality of memories; and

 a three-dimensional bussed packet routing
20 network including a plurality of communications buses connecting said processing elements; and

a local allocation means for accepting a plurality of outputs from said assembling means;
25 a second input means coupled to said data-flow processing means in order to communicate a plurality of data to said data-flow processing means; and
a plurality of output means coupled to said data-flow processing means in order to communicate a
30 plurality of results from said data-flow processing means to an output terminal means.

1 5. Data-flow apparatus for highly efficient data and signal processing as described in Claim 3 in which each said processing element further comprises:
a plurality of communication means for
5 transmission or reception of digital signals;
a communication part which includes a plurality of queues, a plurality of connections between said queues, and a memory connected to one of said queues;
a processor part which includes a plurality
10 of micromachines, a plurality of queues, a plurality of memories, and a plurality of connections between said micromachines, said memories, and said queues;
a plurality of memories connected so as to receive addresses from said processor part and to supply
15 to or receive data from said processor part; and
a bus connecting said communication part to said processor part.

1 6. Data-flow apparatus for highly efficient data and signal processing as described in Claim 5 in which each said three-dimensional bussed packet routing network further includes a bidirectional plane bus, a
5 bidirectional column bus, and a bidirectional row bus, and in which each said processing element is connected to said bidirectional plane, column, and row buses.

- 1 7. Data-flow apparatus for highly efficient data
and signal processing as described in Claim 6 in which
each said communication part of each said processing
element further includes:
- 5 a bidirectional processor bus;
 a first-in-first-out plane input buffer queue,
connected to said plane bus;
 a first-in-first-out plane output buffer
queue, connected to said plane bus;
- 10 a first-in-first-out column input buffer
queue, connected to said column bus;
 a first-in-first-out column output buffer
queue, connected to said column bus;
- 15 a first-in-first-out row input buffer queue,
connected to said row bus;
 a first-in-first-out row output buffer queue,
connected to said row bus;
- 20 a first-in-first-out processor input buffer
queue, connected to said processor bus;
 a first-in-first-out processor output buffer
queue, connected to said processor bus;
- 25 a first internal bus connected to said
processor input, plane input, column input, and row
input buffer queues for sending packets from said
processor, plane, column, and row input buffer queues
to said processor, plane, column, and row output queues.
- 30 a second internal bus connected to said
processor output, plane output, column output, and row
output buffer queue for sending packets from said
processor, plane, column, and row input buffer queues
to said processor, plane, column, and row output buffer
queues;

an error memory; and
a bidirectional error memory bus connecting
35 said error memory to said processor input buffer queue;
and each said processor part of each said processing
element further includes:
a template memory controller micromachine;
a fire detect memory, forming part of said
40 template memory controller micromachine;
an arithmetic and logic unit (ALU) micromachine;
a microprocessor, forming part of said ALU
micromachine;
a micromemory, forming part of said ALU
45 micromachine which controls said ALU micromachine;
a destination tagger micromachine;
a destination memory controller micromachine;
a template memory, connected to said
template memory controller micromachine so as to receive
50 addresses from said template memory controller micromachine
and to receive data from or supply data to said template
memory controller micromachine;
a first-in-first-out firing queue, connected
from said template memory controller micromachine to
55 said ALU micromachine;
a first-in-first-out result queue connected
from said ALU micromachine to said destination tagger
micromachine;
a bidirectional controller bus linking said
60 destination memory controller micromachine to said
template memory controller micromachine;
a first-in-first-out feedback queue connected
from said destination tagger micromachine to said
bidirectional controller bus;
65 a first-in-first-out "to communication" queue
connected from said destination tagger micromachine to
said bidirectional processor bus;

a first-in-first-out "from communication" queue connected from said bidirectional processor bus to said bidirectional controller bus;
70 a first-in-first-out associated information queue;
a first-in-first-out destination queue;
a destination memory, connected to said
75 destination memory controller micromachine so as to receive addresses from said destination memory controller micromachine; and
a bidirectional destination memory data bus connected to said destination memory and linking said
80 destination memory with said destination queue, said associated information queue, and said ALU micromachine, to communicate data between said destination memory and said destination queue, said associated information queue, and said ALU micromachine.

1 8. Data-flow apparatus for highly efficient data and signal processing as described in Claim 7 in which said processor part of said processing element and said communication part of said processing element are
5 both implemented in very large scale integration (VLSI) circuitry.

1 9. Data-flow apparatus for highly efficient data and signal processing as described in Claim 4 in which each said processing element further includes:
a plurality of communication means for
5 transmission and reception of digital signals;
a communication part which includes a plurality of queue, a plurality of connections between said queues, and a memory connected to one of said queues;

- 10 a processor part which includes a plurality
of micromachines, a plurality of queues, a plurality of
memories, and a plurality of connections between said
micromachines, said memories, and said queues;
15 a plurality of memories connected so as to
receive addresses from said processor part and to supply
to or receive data from said processor part; and
 a bus connecting said communication part to
said processor part.

- 1 10. Data-flow apparatus for highly efficient data
and signal processing as described in Claim 5 in which
each said three-dimensional bussed packet routing
network further includes a bidirectional plane bus, a
5 bidirectional column bus, and a bidirectional row bus,
and in which each said processing element is connected
to said bidirectional plane, column, and row buses.

- 1 11. Data-flow apparatus for highly efficient data
and signal processing as described in Claim 10 in which
each said communication part of each said processing
element further includes:
5 a bidirectional processor bus;
 a first-in-first-out plane input buffer
queue, connected to said plane bus;
 a first-in-first-out plane output buffer
queue, connected to said plane bus;
10 a first-in-first-out column input buffer
queue, connected to said column bus;
 a first-in-first-out column output buffer
queue, connected to said column bus;
 a first-in-first-out row input buffer queue,
15 connected to said row bus;
 a first-in-first-out row output buffer queue,
connected to said row bus;

a first-in-first-out processor input buffer queue, connected to said processor bus;

20 a first-in-first-out processor output buffer queue, connected to said processor bus;

a first internal bus connected to said processor input, plane input, column input, and row input buffer queues for sending packets from said processor, plane, column, and row input buffer queues

25 to said processor, plane, column, and row output queues;

a second internal bus connected to said processor output, plane output, column output, and row output buffer queue for sending packets from said processor, plane, column, and row input buffer queues

30 to said processor, plane, column, and row output buffer queues;

an error memory; and

a bidirectional error memory bus connecting

35 said error memory to said processor input buffer queue; and each said processor part of each said processing element further includes:

a template memory controller micromachine;

a fire detect memory, forming part of said

40 template memory controller micromachine;

an arithmetic and logic unit (ALU) micromachine;

a microprocessor, forming part of said ALU micromachine;

a micromemory, forming part of said ALU

45 micromachine, which controls said ALU micromachine;

a destination tagger micromachine;

a destination memory controller micromachine;

a template memory, connected to said template memory controller micromachine so as to receive addresses

50 from said template memory controller micromachine and to receive data from or supply data to said template memory controller micromachine;

BAD ORIGINAL

a first-in-first-out firing queue, connected from said template memory controller micromachine to said ALU micromachine;

55 a first-in-first-out result queue connected from said ALU micromachine to said destination tagger micromachine;

a bidirectional controller bus linking said destination memory controller micromachine to said template memory controller micromachine;

60 a first-in-first-out feedback queue connected from said destination tagger micromachine to said bidirectional controller bus;

65 a first-in-first-out "to communication" queue connected from said destination tagger micromachine to said bidirectional processor bus;

a first-in-first-out "from communication" queue connected from said bidirectional processor bus to said bidirectional controller bus;

70 a first-in-first-out associated information queue;

a first-in-first-out destination queue;

a destination memory, connected to said destination memory controller micromachine so as to receive addresses from said destination memory controller micromachine; and

75 a bidirectional destination memory data bus connected to said destination memory and linking said destination memory with said destination queue, said associated information queue, and said ALU micromachine,

80 to communicate data between said destination memory and said destination queue, said associated information queue, and said ALU micromachine.

- 1 12. Data-flow apparatus for highly efficient data
and signal processing as described in Claim 11 in which
said processor part of said processing element and said
communication part of said processing element are both
5 implemented in very large scale integration (VLSI)
circuitry.

1/10

Fig. 1.

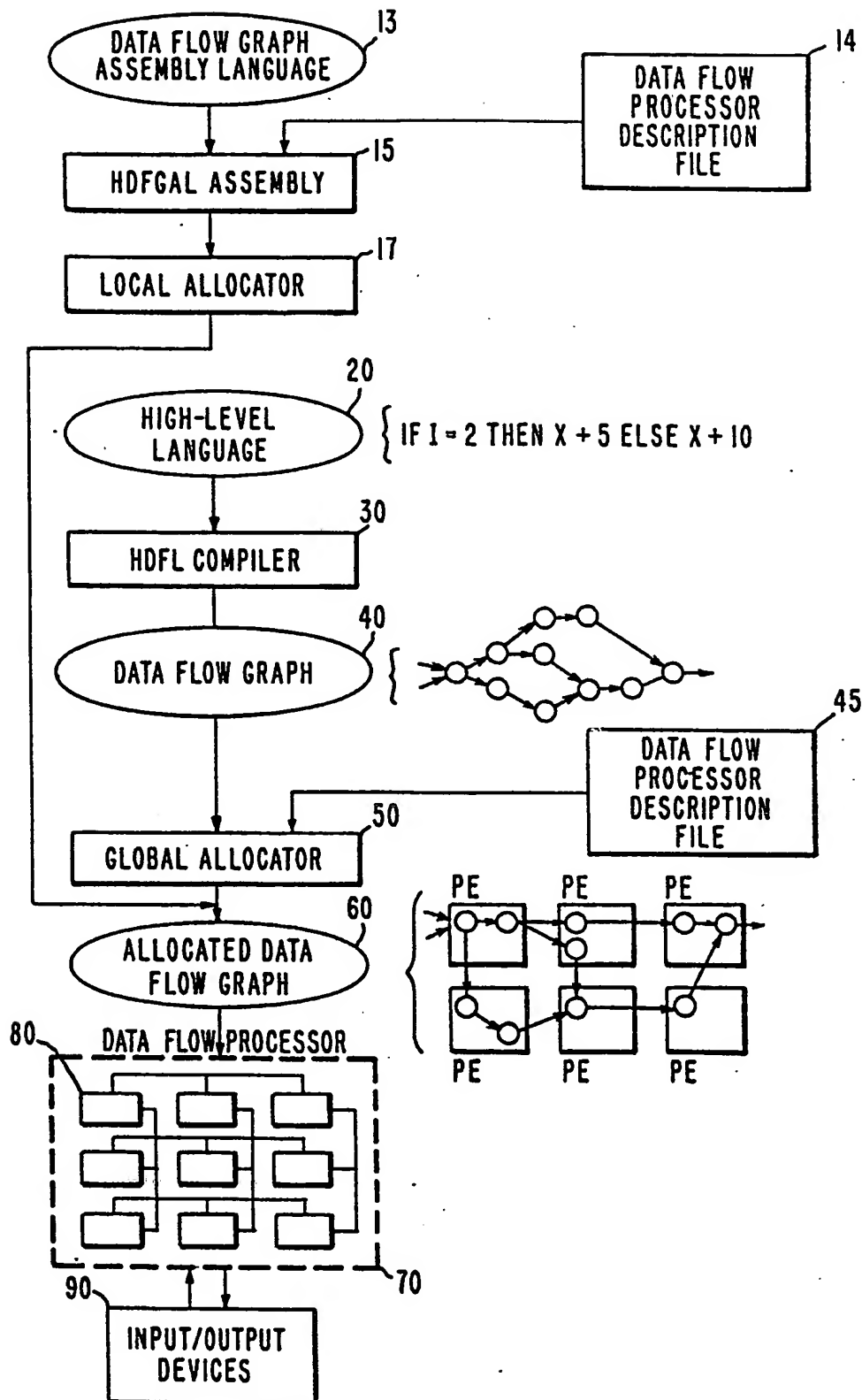


Fig. 2.

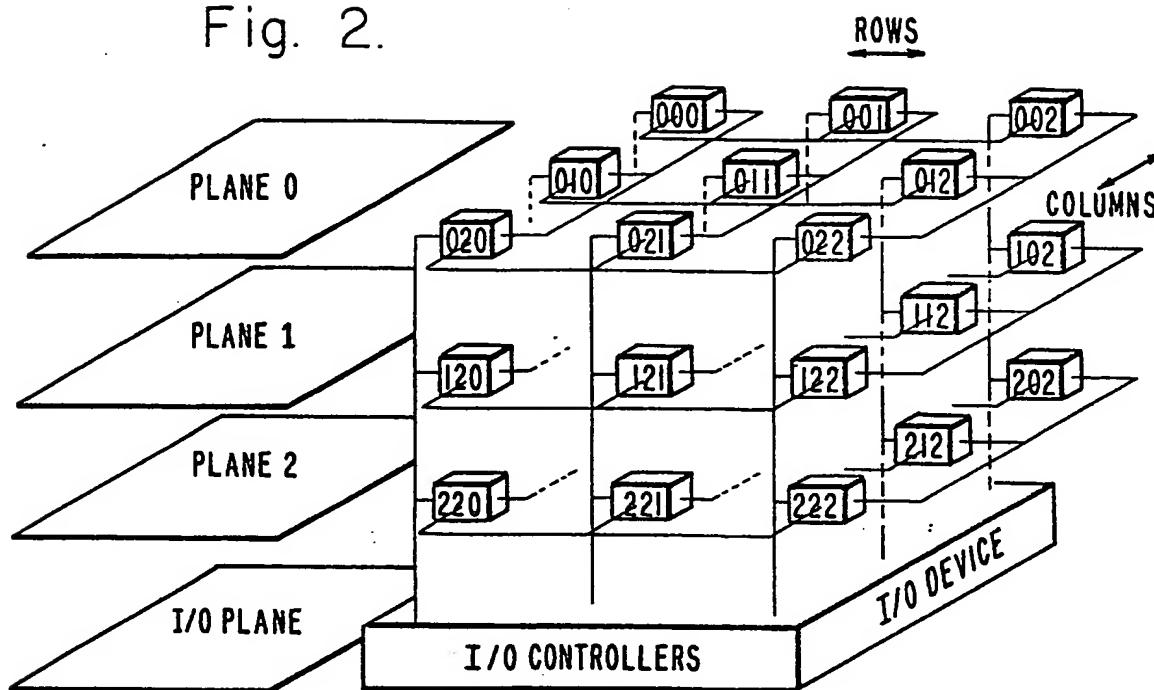
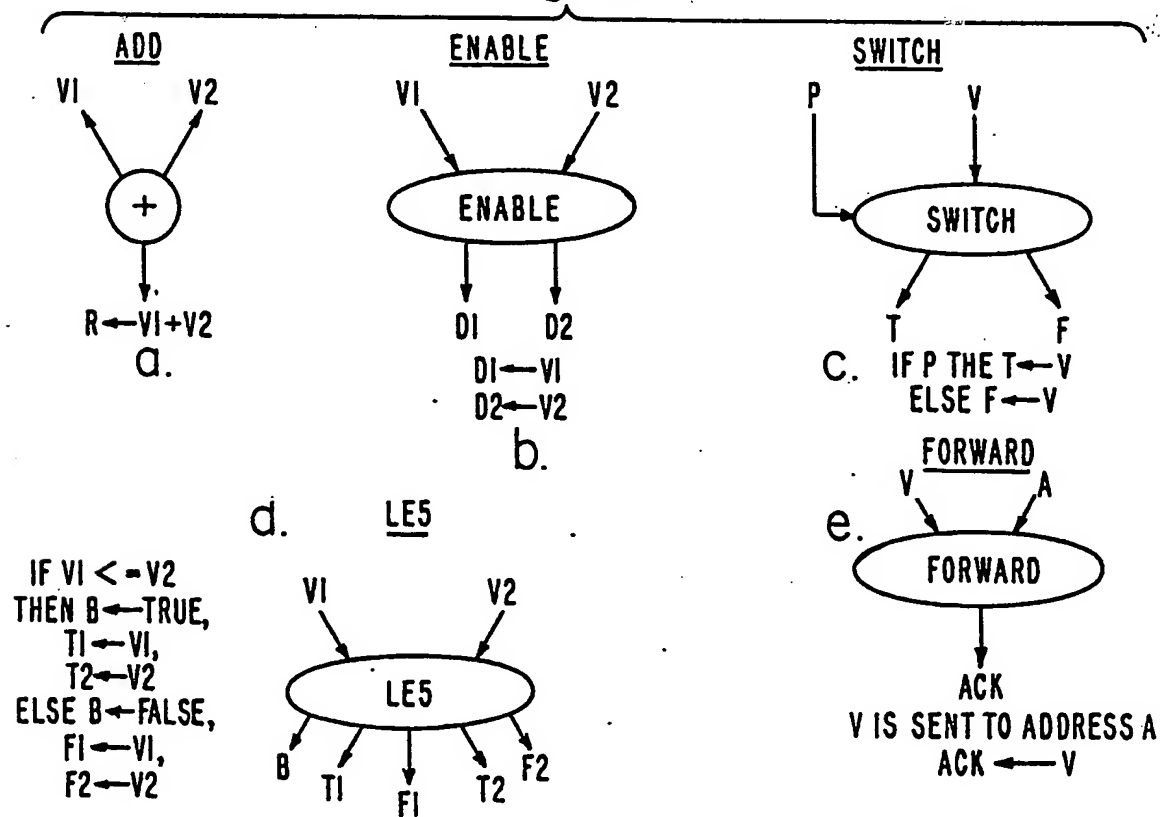


Fig. 6.



3/10

Fig. 3.

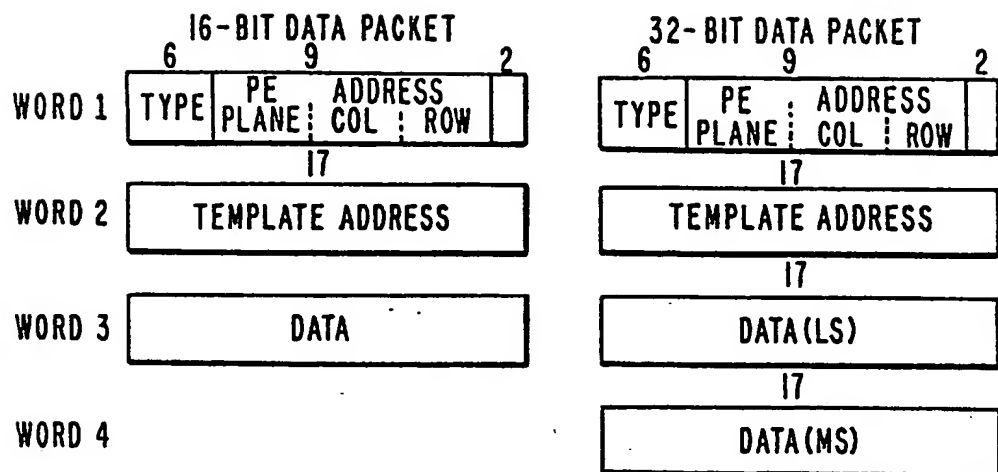


Fig. 5.

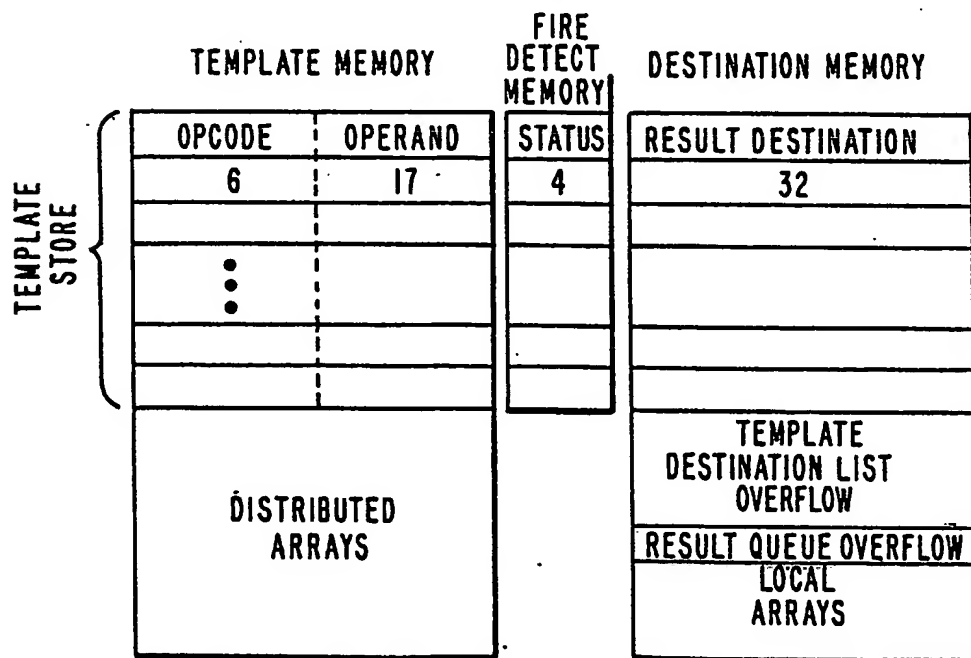


Fig. 4.

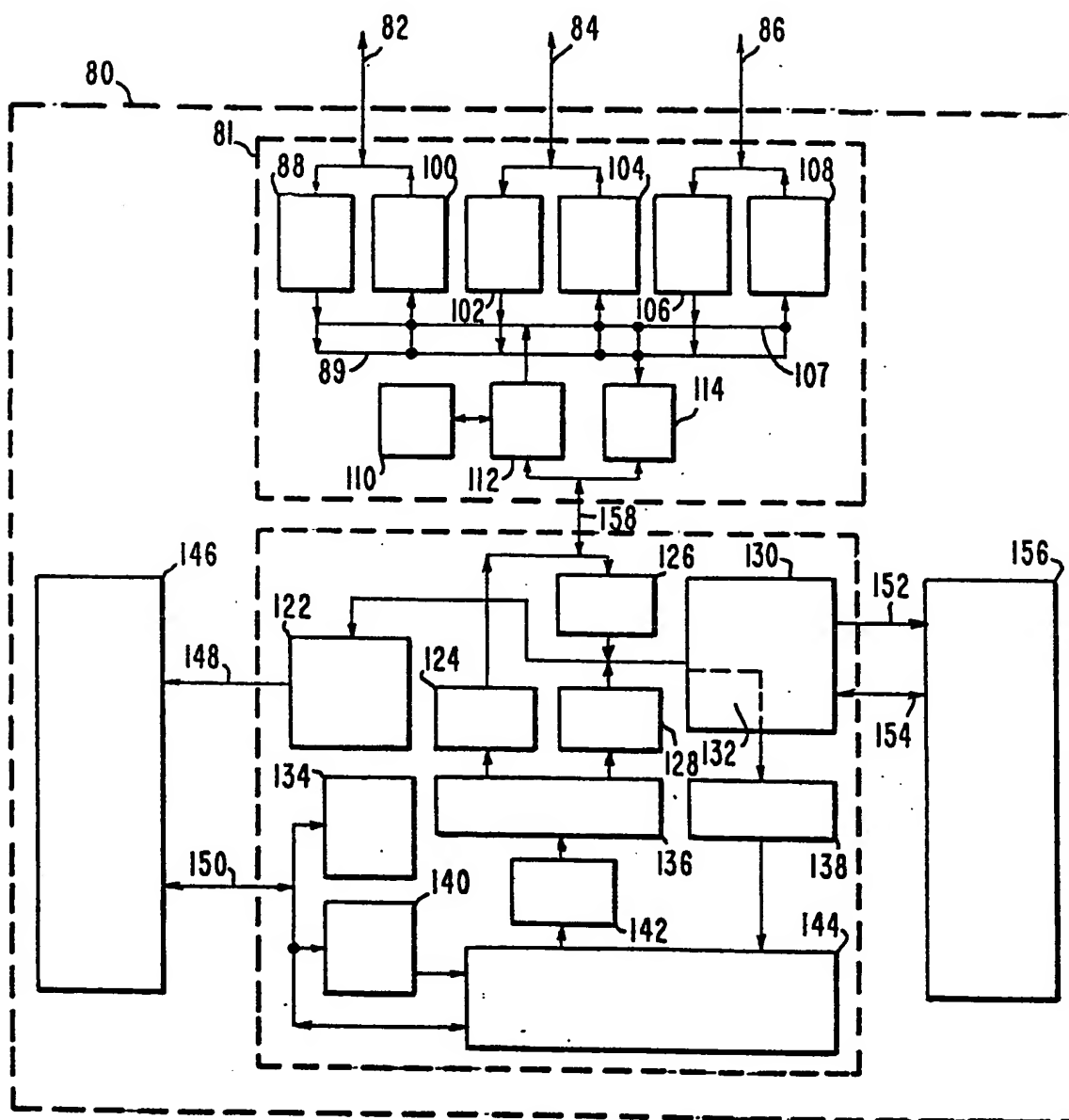
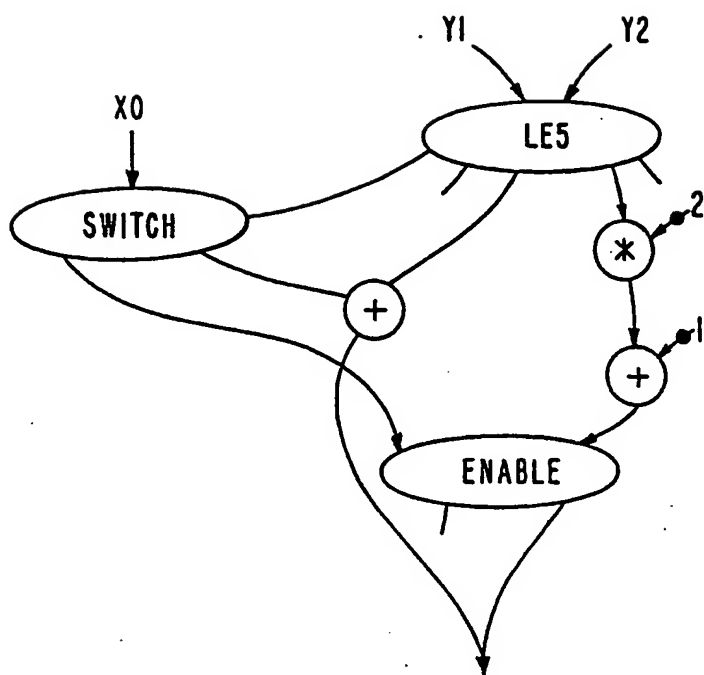


Fig. 7.



6/10

Fig. 9.

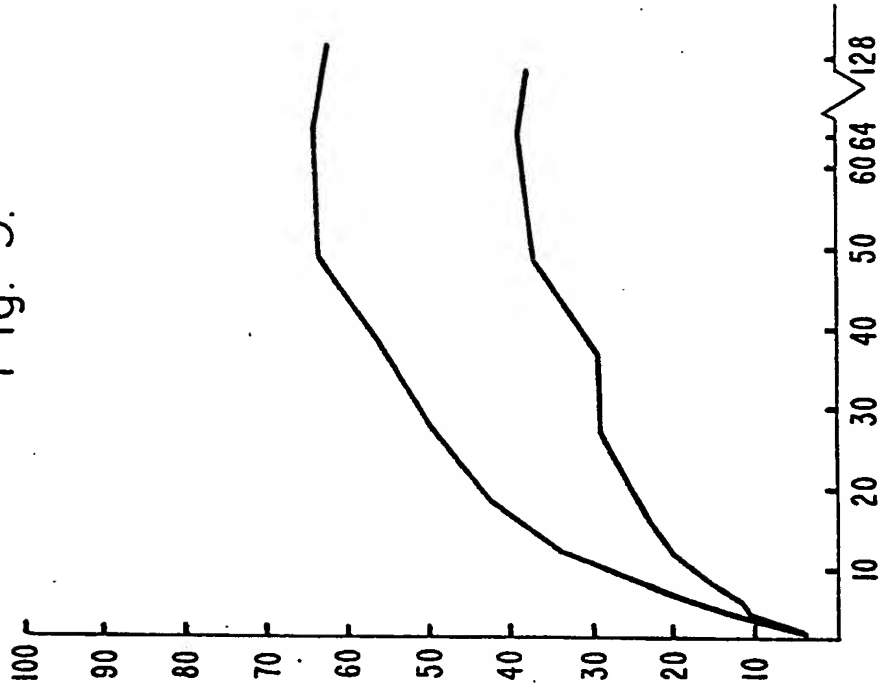
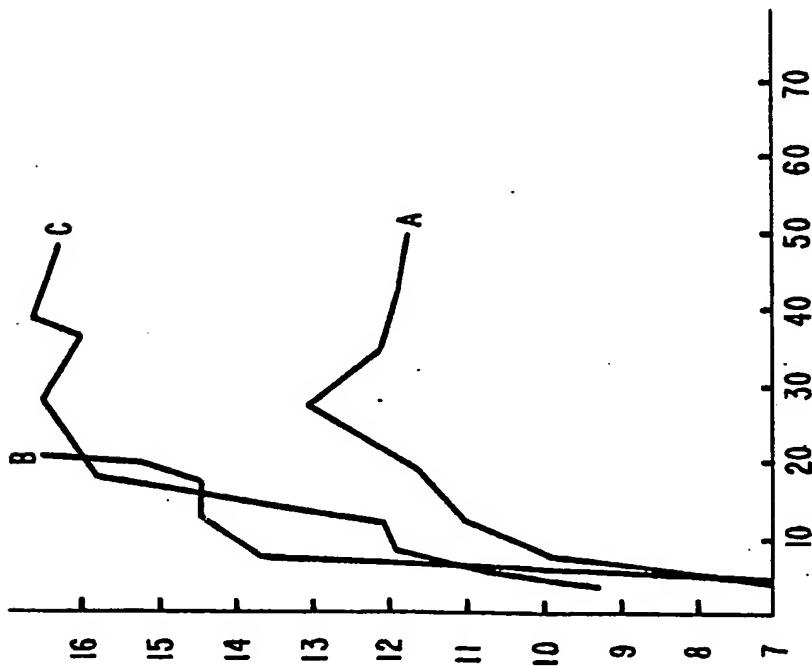


Fig. 8.



7/10

Fig. 11.

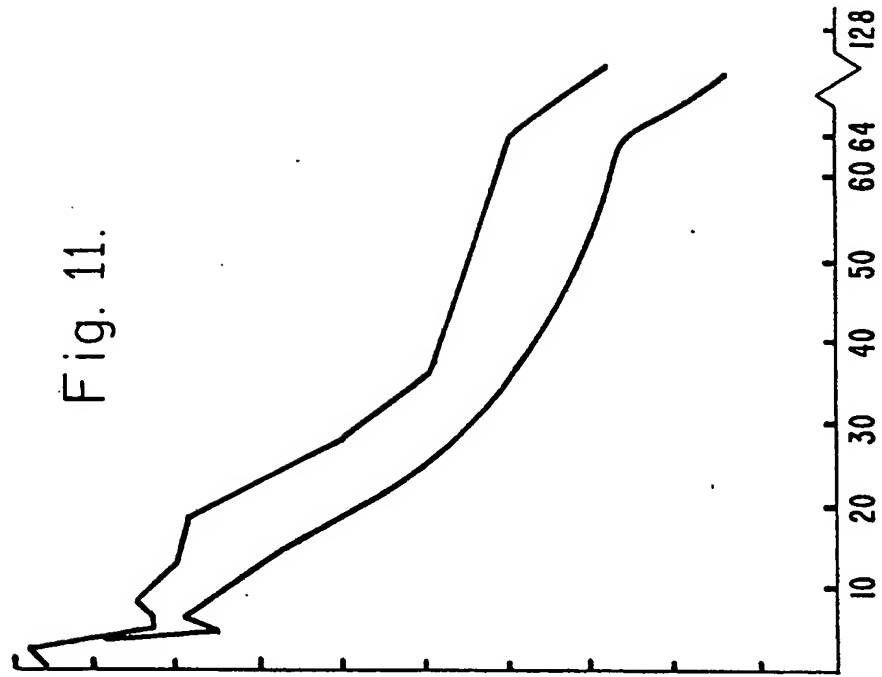
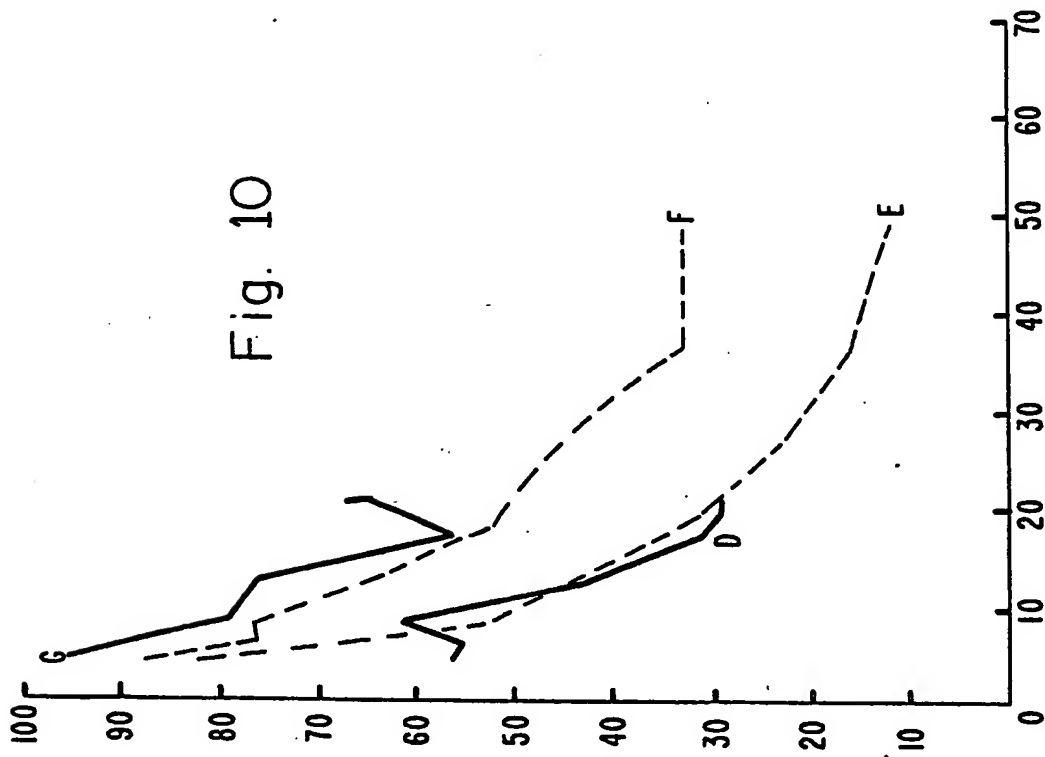


Fig. 10



8/10

Fig. 13.

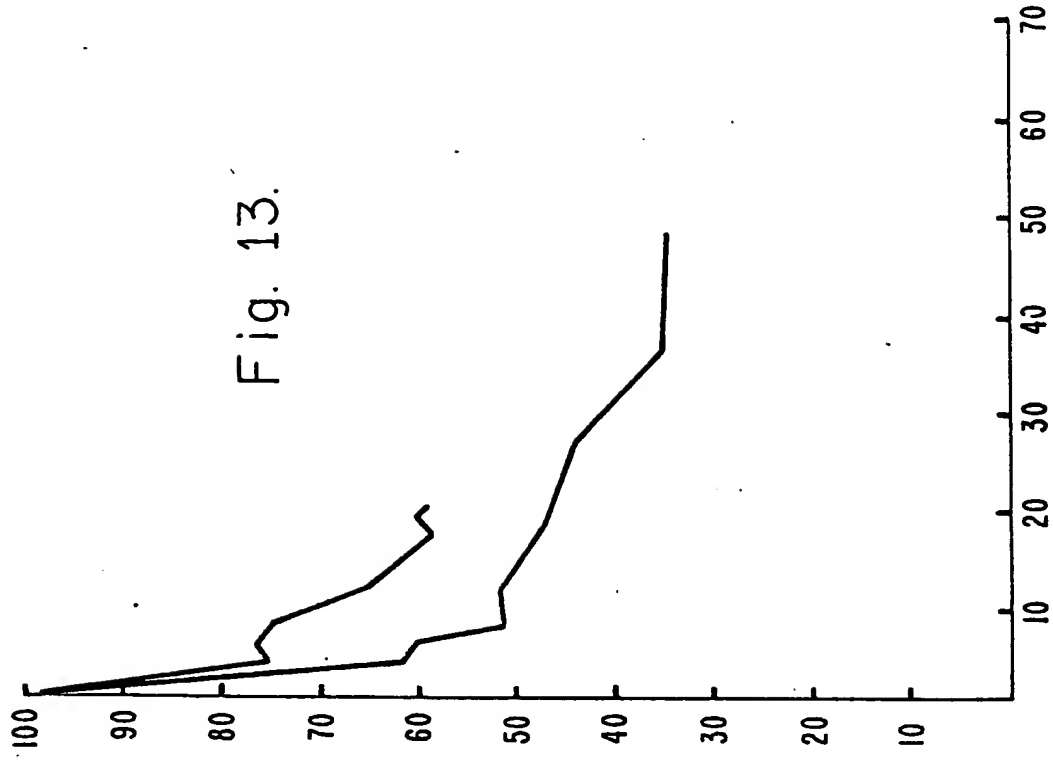


Fig. 12.

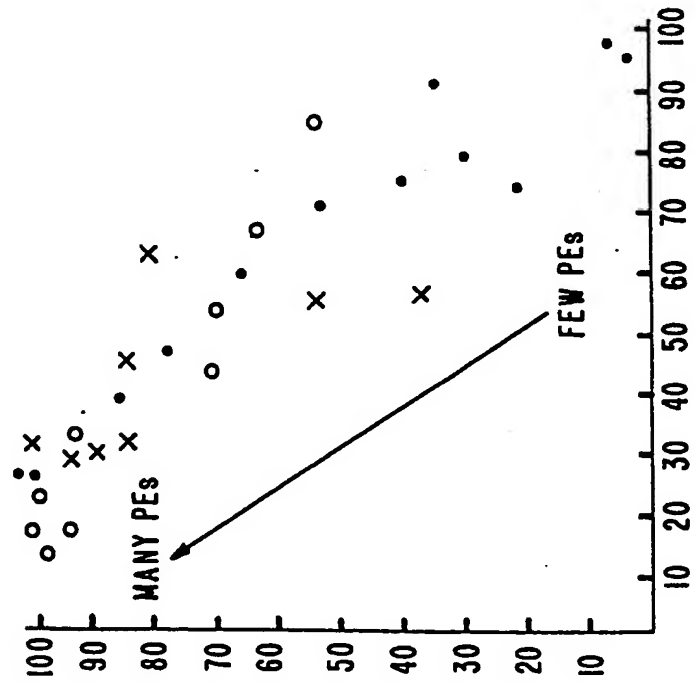


Fig. 15.

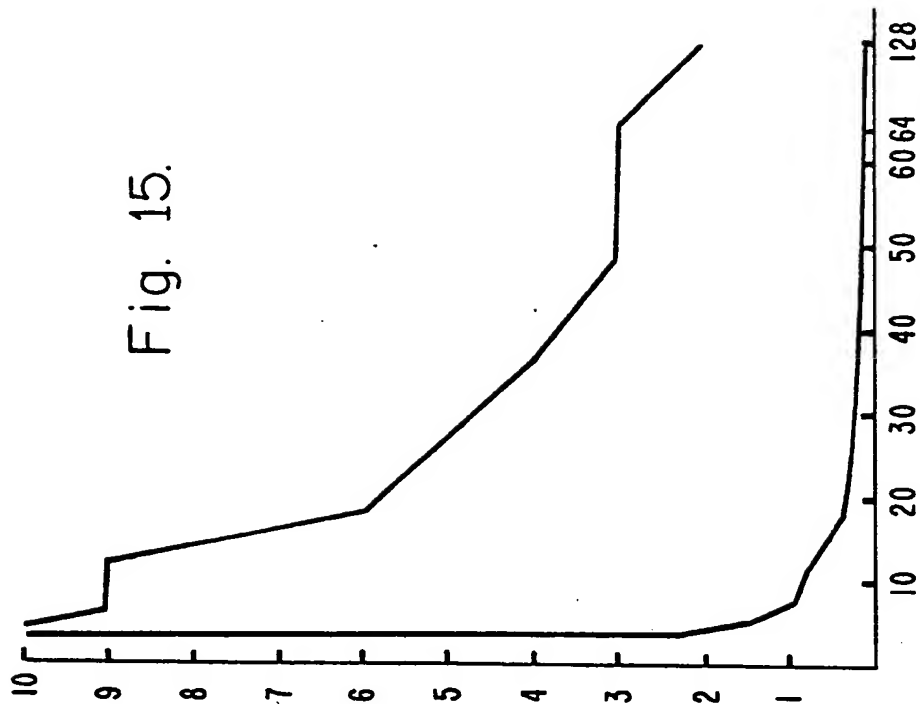
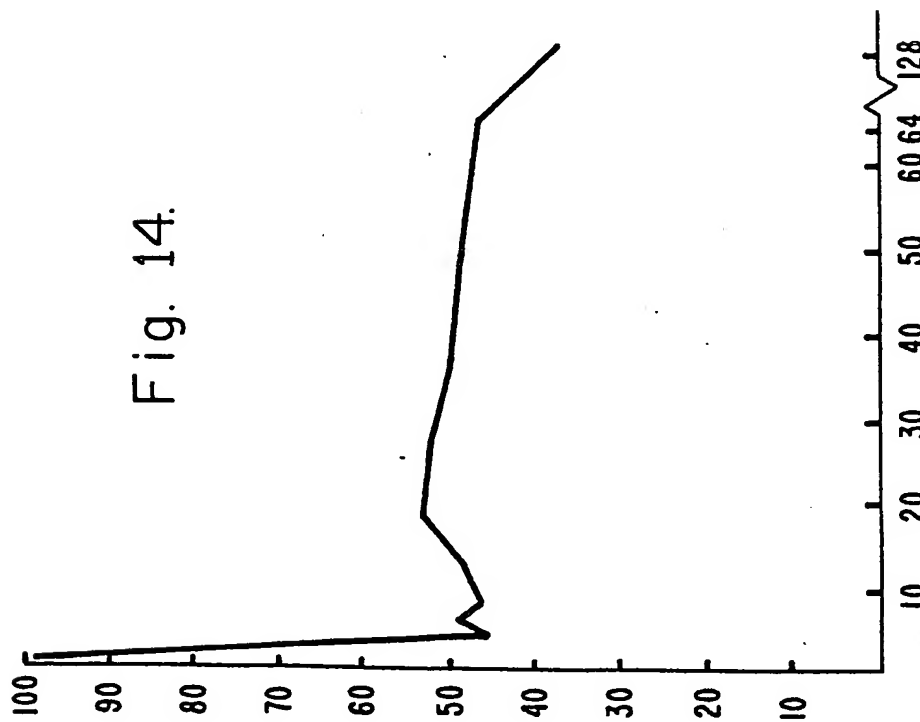
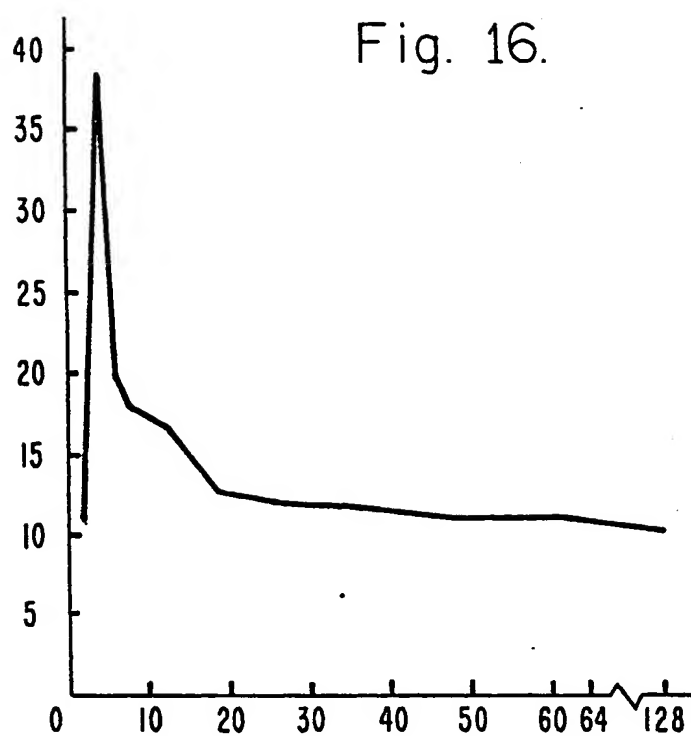


Fig. 14.



10/10

Fig. 16.



INTERNATIONAL SEARCH REPORT

International Application No PCT/US 87/00410

I. CLASSIFICATION OF SUBJECT MATTER (if several classification symbols apply, indicate all) ⁶ According to International Patent Classification (IPC) or to both National Classification and IPC IPC ⁴ : G 06 F 9/44														
II. FIELDS SEARCHED <div style="text-align: center; border-top: 1px solid black; border-bottom: 1px solid black; margin: 5px 0;">Minimum Documentation Searched ⁷</div> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 25%; border-bottom: 1px solid black; padding: 5px;">Classification System</td> <td style="border-bottom: 1px solid black; padding: 5px;">Classification Symbols</td> </tr> <tr> <td style="padding: 5px;">IPC⁴</td> <td style="padding: 5px;">G 06 F 9</td> </tr> </table> <div style="text-align: center; border-top: 1px solid black; border-bottom: 1px solid black; margin: 5px 0;">Documentation Searched other than Minimum Documentation to the Extent that such Documents are Included in the Fields Searched ⁸</div>			Classification System	Classification Symbols	IPC ⁴	G 06 F 9								
Classification System	Classification Symbols													
IPC ⁴	G 06 F 9													
III. DOCUMENTS CONSIDERED TO BE RELEVANT ⁹ <table style="width: 100%; border-collapse: collapse;"> <tr> <th style="width: 10%; border-bottom: 1px solid black; padding: 5px;">Category ⁹</th> <th style="width: 60%; border-bottom: 1px solid black; padding: 5px;">Citation of Document, ¹¹ with indication, where appropriate, of the relevant passages ¹²</th> <th style="width: 30%; border-bottom: 1px solid black; padding: 5px;">Relevant to Claim No. ¹³</th> </tr> <tr> <td style="text-align: center; vertical-align: top; padding: 5px;">X</td> <td style="padding: 5px;">IEEE Transactions on Computers, vol. C-34, no. 12, December 1985 (New York, USA), J.L. Gaudiot et al., "A distributed VLSI architecture for efficient signal and data processing", pages 1072-1087, see the whole document --</td> <td style="text-align: center; vertical-align: top; padding: 5px;">1-12</td> </tr> <tr> <td style="text-align: center; vertical-align: top; padding: 5px;">X</td> <td style="padding: 5px;">The 12th Annual International Symposium on Computer Architecture, 17-19 June 1985, Boston, Massachusetts, USA, (IEEE Computer Society Press, USA), R. Vedder et al., "The Hughes data flow multiprocessor: architecture for efficient signal and data processing", pages 324-332, see the whole document --</td> <td style="text-align: center; vertical-align: top; padding: 5px;">1-12</td> </tr> <tr> <td style="text-align: center; vertical-align: top; padding: 5px;">X</td> <td style="padding: 5px;">The 5th International Conference on Distributed Computing Systems, 13-17 May 1985, Denver, Colorado, USA, (IEEE Computer Society Press, USA), R. Vedder et al., "The Hughes data flow multiprocessor", pages 2-9, see the whole document --</td> <td style="text-align: center; vertical-align: top; padding: 5px;">1-12 ./.</td> </tr> </table>			Category ⁹	Citation of Document, ¹¹ with indication, where appropriate, of the relevant passages ¹²	Relevant to Claim No. ¹³	X	IEEE Transactions on Computers, vol. C-34, no. 12, December 1985 (New York, USA), J.L. Gaudiot et al., "A distributed VLSI architecture for efficient signal and data processing", pages 1072-1087, see the whole document --	1-12	X	The 12th Annual International Symposium on Computer Architecture, 17-19 June 1985, Boston, Massachusetts, USA, (IEEE Computer Society Press, USA), R. Vedder et al., "The Hughes data flow multiprocessor: architecture for efficient signal and data processing", pages 324-332, see the whole document --	1-12	X	The 5th International Conference on Distributed Computing Systems, 13-17 May 1985, Denver, Colorado, USA, (IEEE Computer Society Press, USA), R. Vedder et al., "The Hughes data flow multiprocessor", pages 2-9, see the whole document --	1-12 ./.
Category ⁹	Citation of Document, ¹¹ with indication, where appropriate, of the relevant passages ¹²	Relevant to Claim No. ¹³												
X	IEEE Transactions on Computers, vol. C-34, no. 12, December 1985 (New York, USA), J.L. Gaudiot et al., "A distributed VLSI architecture for efficient signal and data processing", pages 1072-1087, see the whole document --	1-12												
X	The 12th Annual International Symposium on Computer Architecture, 17-19 June 1985, Boston, Massachusetts, USA, (IEEE Computer Society Press, USA), R. Vedder et al., "The Hughes data flow multiprocessor: architecture for efficient signal and data processing", pages 324-332, see the whole document --	1-12												
X	The 5th International Conference on Distributed Computing Systems, 13-17 May 1985, Denver, Colorado, USA, (IEEE Computer Society Press, USA), R. Vedder et al., "The Hughes data flow multiprocessor", pages 2-9, see the whole document --	1-12 ./.												
<div style="display: flex; justify-content: space-between;"> <div style="width: 45%;"> <p>¹⁰ Special categories of cited documents:</p> <p>"A" document defining the general state of the art which is not considered to be of particular relevance</p> <p>"E" earlier document but published on or after the international filing date</p> <p>"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)</p> <p>"O" document referring to an oral disclosure, use, exhibition or other means</p> <p>"P" document published prior to the international filing date but later than the priority date claimed</p> </div> <div style="width: 45%;"> <p>"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention</p> <p>"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step</p> <p>"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art</p> <p>"A" document member of the same patent family</p> </div> </div>														
IV. CERTIFICATION <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%; border-bottom: 1px solid black; padding: 5px;">Date of the Actual Completion of the International Search</td> <td style="width: 50%; border-bottom: 1px solid black; padding: 5px;">Date of Mailing of this International Search Report</td> </tr> <tr> <td style="padding: 5px;">22nd June 1987</td> <td style="padding: 5px;">28 JUL 1987</td> </tr> <tr> <td style="border-bottom: 1px solid black; padding: 5px;">International Searching Authority</td> <td style="border-bottom: 1px solid black; padding: 5px;">Signature of Authorized Officer</td> </tr> <tr> <td style="padding: 5px;">EUROPEAN PATENT OFFICE</td> <td style="padding: 5px;">M. VAN MOI </td> </tr> </table>			Date of the Actual Completion of the International Search	Date of Mailing of this International Search Report	22nd June 1987	28 JUL 1987	International Searching Authority	Signature of Authorized Officer	EUROPEAN PATENT OFFICE	M. VAN MOI				
Date of the Actual Completion of the International Search	Date of Mailing of this International Search Report													
22nd June 1987	28 JUL 1987													
International Searching Authority	Signature of Authorized Officer													
EUROPEAN PATENT OFFICE	M. VAN MOI													

International Application No. PCT/US 87/00410

III. DOCUMENTS CONSIDERED TO BE RELEVANT (CONTINUED FROM THE SECOND SHEET)		
Category *	Citation of Document, with indication, where appropriate, of the relevant passages	Relevant to Claim No
X	Proceedings of the 1985 International Conference on Parallel Processing, 20-23 August 1985, Washington, USA, (IEEE Computer Society Press, USA), M.L. Campbell, "Static allocation for a data flow multiprocessor", pages 511- 517, see the whole document	1-4 5-12
A	-----	